

On Throughput-Delay Optimal Access to Storage Clouds via Load Adaptive Coding and Chunking

Guanfeng Liang, *Member, IEEE*, and Ulaş C. Kozat, *Senior Member, IEEE*

Abstract—Recent literature including our past work provide analysis and solutions for using (i) erasure coding, (ii) parallelism, or (iii) variable slicing/chunking (i.e., dividing an object of a specific size into a variable number of smaller chunks) in speeding up the I/O performance of storage clouds. However, a comprehensive approach that considers all three dimensions together to achieve the best throughput-delay trade-off curve had been lacking. This paper presents the first set of solutions that can pick the best combination of coding rate and object chunking/slicing options as the load dynamically changes. Our specific contributions are as follows: (1) We establish via measurements that combining variable coding rate and chunking is mostly feasible over a popular public cloud. (2) We relate the delay optimal values for chunking level and code rate to the queue backlogs via an approximate queuing analysis. (3) Based on this analysis, we propose TOFEC that adapts the chunking level and coding rate against the queue backlogs. Our trace-driven simulation results show that TOFEC’s adaptation mechanism converges to an appropriate code that provides the optimal throughput-delay trade-off without reducing system capacity. Compared to a non-adaptive strategy optimized for throughput, TOFEC delivers $2.5\times$ lower latency under light workloads; compared to a non-adaptive strategy optimized for latency, TOFEC can scale to support over $3\times$ as many requests. (4) We propose a simpler greedy solution that performs on a par with TOFEC in average delay performance, but exhibits significantly more performance variations.

Index Terms—FEC, Cloud storage, Queueing, Delay

I. INTRODUCTION

Cloud storage has gained wide adoption as an economic, scalable, and reliable mean of providing data storage tier for applications and services. Typical cloud storage systems are implemented as key-value stores in which data objects are stored and retrieved via their unique keys. To provide high degree of availability, scalability, and data durability, each object is replicated several times within the internal distributed file system and sometimes also further protected by erasure codes to more efficiently use the storage capacity while attaining very high durability guarantees [1].

Cloud storage providers usually implement a variety of optimization mechanisms such as load balancing and caching/prefetching internally to improve performance. Despite all such efforts, still evaluations of large scale systems indicate that there is a high degree of randomness in delay performance [2]. Thus, services that require more robust and predictable Quality of Service (QoS) must deploy their own external solutions such as sending multiple/redundant

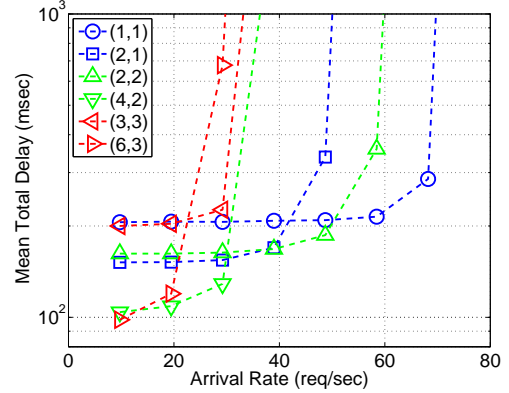


Fig. 1. Delay for downloading 3MB files using fixed MDS codes

requests (in parallel or sequentially), chunking large objects into smaller ones and read/write each chunk through parallel connections, replicate the same object using multiple distinct keys in a coded or uncoded fashion, etc.

In this paper, we present **black box** solutions¹ that can provide much better throughput-delay performance for reading and writing files on cloud storage utilizing (i) parallelism, (ii) erasure coding, and (iii) chunking. To the best of our knowledge, our work is the first one that adaptively picks the best erasure coding rate and chunk size to minimize the expected latency without sacrificing the supportable rate region (i.e., maximum requests per second) of the storage tier. The presented solutions can be deployed over a proxy tier external to the cloud storage tier or can be utilized internally by the cloud provider to improve the performance of their storage services for all or a subset of their tenants with higher priority.

A. State of the Art

Among the vast amount of research on improving cloud storage system’s delay performance emerged in the past few years, two groups in particular are closely related to our work presented in this paper:

Erasure Coding with Redundant Requests: As proposed by authors of [3], [4], [5], files (or objects) are divided into a *pre-determined* number of k chunks, each of which is $1/k$ the size of the original file, and encoded into $n > k$ of “coded chunks” using an (n, k) Maximum Distance Separable (MDS) code, or more generally a Forward Error Correction (FEC) code. Downloading/uploading of the original file is

G. Liang and U.C. Kozat are with DOCOMO Innovations Inc., Palo Alto, California USA. G. Liang is the contact author. E-mail: gliang@docomoinnovations.com

¹They use only the API provided by storage clouds and do not require any modification or knowledge of the internal implementation of storage clouds.

accomplished by downloading/uploading n coded chunks using parallel connections simultaneously and is deemed served when download/upload of any k coded chunks complete. Such mechanisms significantly improves the delay performance under light workload. However, as shown in our previous work [3] and later reconfirmed by [5], system capacity is reduced due to the overhead for using smaller chunks and redundant requests. This phenomenon is illustrated in Fig.1 where we plot the throughput-delay trade-off for using different MDS codes from our simulations using delays traces collected on Amazon S3. Codes with different k are grouped in different colors. Using a code with high level of chunking and redundancy, in this case a $(6, 3)$ code, although delivers $2\times$ gain in delay at light workload, reduces system capacity to only 30% of the original basic strategy without chunking and redundancy, i.e., $(1, 1)$ code!

This problem is partially addressed in [3] where we present strategies that adjust n according to workload level so that it achieves the near-optimal throughput-delay trade-off for a *predetermined* k . For example, if $k = 3$ is used, the strategies in [3] will achieve the lower-envelope of the red curves in Fig.1. Yet, it still suffers from an almost 60% loss in system capacity.

Dynamic Job Sizing: It has been observed in [2], [6] that in key-value storage systems such as Amazon S3 and Microsoft's Azure Storage, throughput is dramatically higher when they receive a small number of storage access requests for large jobs (or objects) than if they receive a large number of requests for small jobs (or objects), because each storage request incurs overheads such as networking delay, protocol-processing, lock acquisitions, transaction log commits, etc. Authors of [6] developed Stout in which requests are dynamically batched to improve throughput-delay trade-off of key-value storage systems. Based on the observed congestion Stout increase or reduce the batching size. Thus, at high congestion, a larger batch size is used to improve the throughput while at low congestion a smaller batch size is adopted to reduce the delay.

B. Main Contributions

Our work unifies the ideas of redundant requests with erasure coding and dynamic job sizing together in one solution framework. Our major contributions can be listed as follows.

- Providing dynamic job sizing while maintaining parallelism and erasure coding gains is a non-trivial undertaking. Key-value stores map an object key to one or more physical storage nodes (if replication is used). Depending on the implementation, a request for a key might always go to the same physical node or load balanced across all replicas. As detailed in Section III, one has the option of using unique keys for each chunk of an object or share the same key across chunks but assign them different byte ranges. The former wastes significant storage capacity, whereas the latter will likely demonstrate higher correlation across parallel reads/writes of distinct chunks of the same object. Nonetheless, our measurements in different regions over a popular public cloud establish that in fact sharing the same key results in reasonably well

weak-correlations enabling parallelism and coding gains. However, our measurements also indicate that indeed universally good performance is not guaranteed as one region fails to deliver this weak-correlation.

- Exact analysis for computing the optimal code rate and chunking level is far beyond trivial. In Sections IV-A to IV-C, we relate the delay optimal values for chunking level and code rate to the queue backlogs via an approximate queuing analysis.
- Using this analysis, in Section IV-D, we introduce TOFEC (Throughput Optimal FEC Cloud) that implements dynamic adjustment of chunking and redundancy levels to provide the optimal throughput-delay trade-off. In other words, TOFEC achieves the lower envelope of curves in all colors in Fig.1.
- The primary novelty of TOFEC is in its backlog-based adaptive algorithm for dynamically adjusting the chunk size as well as the number of redundant requests issued to fulfill storage access requests. This algorithm of variable chunk sizing can be viewed as a novel integration of prior observations from the two bodies of works discussed above. Based on the observed backlog level as an indicator of the workload, TOFEC increases or decreases the chunk size, as well as the number of redundant requests. In our trace-driven evaluations, we demonstrate that: (1) TOFEC successfully adapts to full range of workloads, delivering $3\times$ lower average delay than the basic static strategy without chunking under light workloads, and under heavy workloads over $3\times$ the throughput of a static strategy with high chunking and redundancy levels optimized for service delay; and (2) TOFEC provides good QoS guarantees as it delivers low delay variations.
- Although TOFEC does not need any explicit information about the internal operations of the storage cloud, it needs to log latency performance and model the cumulative distribution of the delay performance of the storage cloud. We also propose a greedy heuristic that does not need to build such a model, and via trace-driven simulations we show that its performance on average latency is on a par with the performance of TOFEC, but exhibiting significantly higher performance variations.

II. SYSTEM MODELS

A. Basic Architecture and Functionality

The basic system architecture captures how Internet services today utilize public or private storage clouds. The architecture consists of proxy servers in the front-end and a key-value store, referred to as storage cloud, in the back-end. Users interact with the proxy through a high-level API and/or user interfaces. The proxy translates every high-level user request (to read or write a file) into a set of $n \geq 1$ tasks. Each task is essentially a basic storage access operation such as `put`, `get`, `delete`, etc. that will be accomplished using low-level APIs provided by the storage cloud. The proxy maintains a certain number of parallel connections to the storage cloud and each task is executed over one of these connections. After a certain number of tasks are completed successfully, the user

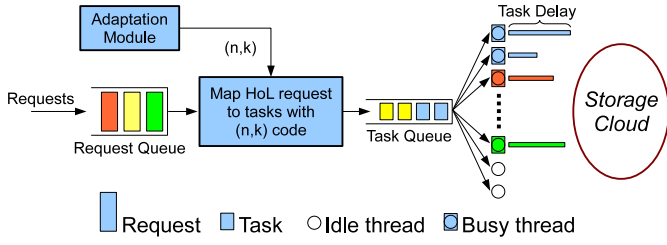


Fig. 2. System Model

request is considered accomplished and the proxy responds to the user with an acknowledgment. The solutions we present are deployed on the proxy server side transparent to the storage cloud.

For read request, we assume that the file is pre-coded into $n^{max} \geq n$ coded chunks with an (n^{max}, k) MDS code and stored on the cloud. Completion of downloading any k coded chunks provides sufficient data to reconstruct the requested file. Thus, the proxy decodes the requested file from the k downloaded chunks and replies to the client. The $n - k$ unfinished and/or not-yet-started tasks are then canceled and removed from the system.

For write request, the file to be uploaded is divided and encoded into n coded chunks using an (n, k) MDS code and hence completion of uploading any k coded chunks means sufficient data have been stored onto the cloud. Thus, upon receiving k successful responses from the storage cloud, the proxy sends a *speculative* success response to the client, without waiting for the remaining $(n - k)$ tasks to finish. Such speculative execution is a commonly practiced optimization technique to reduce client perceived delay in many computer systems such as databases and replicated state machines [7]. Depending on the subsequent read profile on the same file, the proxy can (1) continue serving the remaining tasks till all n tasks finish, or (2) change them to low priority jobs that will be served only when system utilization is low, or (3) cancel them preemptively. The proxy can even (4) run a demon program in the background that generates all n^{max} coded chunks from the already uploaded chunks when the system is not busy.

Accordingly, we model the proxy by the queueing system shown in Fig.2. There are two FIFO (first-in-first-out) queues: (i) the *request queue* that buffers all incoming user requests, and (ii) the *task queue* that is a multi-server queue and holds all tasks waiting to be executed. L threads², representing the set of parallel connections to the storage cloud, are attached to the task queue. The adaptation module of TOFEC monitors the state of the queues and the threads, and decides what coding parameter (n, k) to be used for each request. Without loss of generality, we assume that the head-of-line (HoL) request leaves the request queue only when there is at least one idle thread **and** the task queue is empty. A batch of n tasks are then created for that request and injected into the task queue. As soon as any k tasks complete successfully, the request is considered completed. Such a queue system is work

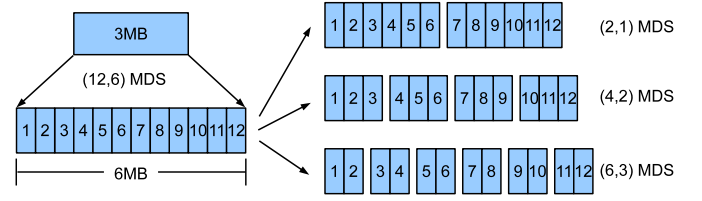


Fig. 3. Example of supporting multiple chunk sizes with Shared Key approach: the 3MB file is divided and encoded into a coded file of 6MB consisting 12 strips, each of 0.5MB. Download the file using a $(2, 1)$ MDS code is accomplished by creating two read tasks: one for strips 1-6, and the other for strips 7-12.

conserving since no thread is left idle as long as there is any request or task pending.

B. Basics of Erasure Codes

An (n, k) MDS code (e.g., Reed-Soloman codes) encodes k data chunks each of B bits into a codeword consisting of n B -bit long coded chunks. The coded chunks can sustain up to $(n - k)$ erasures such that the k original data chunks can be efficiently reconstructed from **any** subset of k coded chunks. n and k are called the length and dimension of the MDS code. We also define $r = n/k$ as the redundancy ratio of an (n, k) MDS code. The erasure resistant property of MDS codes has been utilized in prior works [3], [4], [5], as well as in this paper, to improve delay of cloud storage systems. Essentially a coded chunk experiencing long delay is treated as an erasure.

In this paper, we make use of another interesting property of MDS codes to implement variable chunk sizing of TOFEC in a storage efficient manner: MDS code of high length and dimension for small chunk size can be used as MDS code of smaller code length and dimension for larger chunk size. To be more specific, consider any (N, K) MDS code for chunks of b bits. To avoid confusion, we will refer to these b -bit chunks as strips. A different MDS code of length $n = N/m$, dimension $k = K/m$ and chunk size $B = bm$ for some $m > 1$ can be constructed by simply batching every m data/coded strips into one data/coded chunk. The resulting code is an (n, k) MDS code for B -bit chunks because any k coded chunks covers $mk = K$ coded strips, which is sufficient to reconstruct the original file of $Bk = bm \times K/m = bK$ bits. This property is illustrated as an example in Fig. 3. In this example, a 3MB file is divided into 6 strips of 0.5MB and encoded into 12 coded strips of total size 6MB, using a $(12, 6)$ MDS code. This code can then be used as a $(2, 1)$ code for 3MB chunks, a $(4, 2)$ code for 1.5MB chunks and a $(6, 3)$ code for 1MB chunks **simultaneously** by batching 6, 3 and 2 strips into a chunk.

C. Definitions of Different Delays

The delay experienced by a user request consists of two components: *queueing delay* (D_q) and *service delay* (D_s). Both are defined with respect to the request queue: (i) the queueing delay is the amount of time a request spends waiting in the request queue and (ii) the service delay is the period of time between when the request leaves the request queue (i.e., admitted into the task queue and started being served by at

²We avoid the term “server” that is commonly used in queueing theory literature to prevent confusion.

least one thread) and when it finally leaves the system (i.e., the first time when any k of the corresponding tasks complete). In addition, we also consider the *task delays* (D_t), which is the time it takes for a thread to serve a task assuming it is not terminated or canceled preemptively. To clarify these definitions of delays, consider a request served with an (n, k) MDS code, with T_A being its arrival time, $T_1 \leq T_2 \leq \dots \leq T_n$ the starting times of the corresponding n tasks³. Then the queueing delay is $D_q = T_1 - T_A$. Suppose $D_{t,1}, \dots, D_{t,n}$ are the corresponding task delays, then the completion times of these task will be $X = \{T_1 + D_{t,1}, \dots, T_n + D_{t,n}\}$ if none is canceled. So the request will leave the system at time $X_{(k)}$, which denotes the k -th smallest value in X , i.e., the time when k tasks complete. Then the service delay of this request is $D_s = X_{(k)} - T_1$.

III. VARIABLE CHUNK SIZING

In this section, we discuss implementation issues as well as pros and cons of two potential approaches, namely *Unique Key* and *Shared Key*, for supporting erasure-code-based access to files on the storage cloud with a variety of chunk sizes. Suppose the maximum desired redundancy ratio is r , then these approaches implement variable chunk sizing as follows:

- **Unique Key:** For every choice of chunk size (or equivalently k), a separate batch of rk coded chunks are created and each coded chunk is stored as an individual object with its unique key on the storage cloud. The access to different chunks is implemented through basic `get`, `put` storage cloud APIs.
- **Shared Key:** A coded file is first obtained by stacking together the coded strips obtained by applying a high-dimension ($N = rK, K$) MDS code to the original file, as described in Section II-B and illustrated in Fig.3. For read, the coded file is stored on the cloud as one object. Access to chunks with variable size is realized by downloading segments in the coded file corresponding to batches of a corresponding number of strips, using the same key with more advanced “partial read” storage cloud APIs. Similarly, for write, the file is uploaded in parts using “partial write” APIs and then later merged into one object in the cloud.

A. Implementation and Comparison of the two Approaches

1) *Storage cost:* When the user request is to write a file, storage cost of Unique Key and Shared Key is not so different. However, to support variable chunk sizing for read requests, Shared Key is significantly more cost-efficient than Unique Key. With Shared Key, a single coded file stored on the cloud can be reused to support essentially an arbitrary number of different chunk sizes, as long as the strip size is small enough. On the other hand, it seems impossible to achieve similar reusing with the Unique Key approach where different chunks of the same file is treated as individual objects. So with Unique Key, every additional chunk size to be supported requires an extra storage cost $r \times$ file size. Such linear growth of storage

cost easily makes it prohibitively expensive even to support a small number of chunk sizes.

2) *Diversity in delays:* The success of TOFEC and other proposals to use redundant requests (either with erasure coding or replication) for delay improvement relies on diversity in cloud storage access delays. In particular, TOFEC, as well as [3], [4], [5], requires access delays for different chunks of **the same file** to be weakly correlated.

With Unique Key, since different chunks are treated as individual objects, there is no inherent connection among them from the storage cloud system’s perspective. So depending on the internal implementation of object placement policy of the storage cloud system, chunks of a file can be stored on the cloud in different storage units (disks or servers) on the same rack, or in different racks in the same data center, or even to different data centers at distant geographical locations. Hence it is quite likely that delays for accessing different chunks of the same file show very weak correlation.

On the other hand, with Shared Key, since coded chunks are combined into one coded file and stored as one object in the cloud, it is very likely that the whole coded file, hence all coded chunks/strips, is stored in the same storage unit, unless the storage cloud system internally divides the coded file into pieces and distributes them to different units. Although many distributed storage systems do divide files into parts and store them separately, it is normally only for larger files. For example, the popular Hadoop distributed file system by default does not divide files smaller than 64MB. When different chunks are stored on the same storage unit, we can expect higher correlation in their access delays. It then is to be verified that the correlation between different chunks with the Shared Key approach is still weak enough for our coding solution to be beneficial.

3) *Universal support:* Unique Key is the approach adopted in our previous work [3] to support erasure-code based file accessing with **one predetermined** chunk size. A benefit of Unique Key is that it only requires basic `get` and `put` APIs that all storage cloud systems must provide. So it is readily supported by all storage cloud systems and can be implemented on top of any one.

On the other hand, Shared Key requires more advanced APIs that allow the proxy to download or upload only the targeted segment of an object. Such advanced APIs are not currently supported by all storage cloud systems. For example, to the best of our knowledge currently Microsoft’s Azure Storage provides only methods for “partial read”⁴ but none for “partial write”. On the contrary, Amazon S3 provides partial access for both read and write: the proxy can download a specific inclusive byte range within an object stored on S3 by calling `getObject(request, destination)`⁵; and for uploading an `uploadPart` method to upload segments of an object and an `completeMultipartUpload` method to merge the uploaded segments are provided. We expect more

⁴E.g. `DownloadRangeToStream(target, offset, length)` downloads a segment of `length` bytes starting from the `offset`-th byte of the target object (or “blob” in Azure’s jargon).

⁵The byte range is set by calling `request.setRange(start, end)`.

³We assume $T_i = \infty$ if the i -th task is never started.

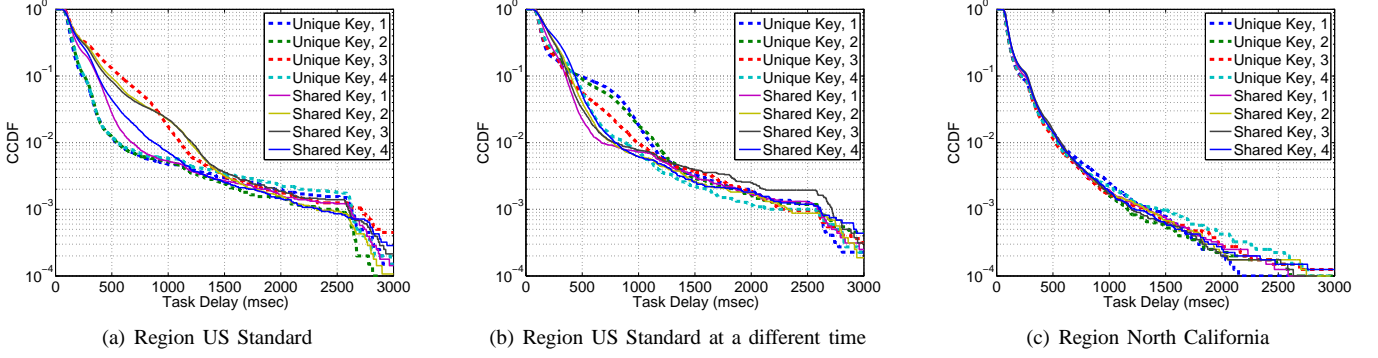


Fig. 4. CCDF of individual threads with 1MB chunks and $n = 4$, measured on May 1st, 2013

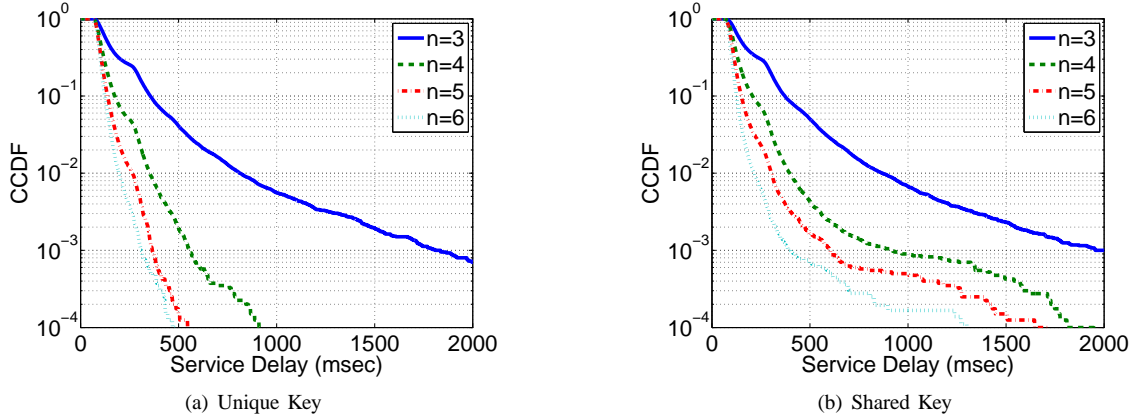


Fig. 5. CCDF of service delay for reading 3MB files with 1MB chunks

service providers to introduce both partial read and write APIs in the near future.

B. Measurements on Amazon S3

To understand the trade-off between Unique Key and Shared Key, we run measurements over Amazon EC2 and S3. EC2 instance served as the proxy in our system model. We instantiated an extra large EC2 instance with high I/O capability in the same availability region as the S3 bucket that stores our objects. We conducted experiments on different week days in May to July 2013 with various chunk sizes between 0.5MB to 3MB and up to $n = 12$ coded chunks per file. For each value of n , we allow $L = n$ simultaneously active threads while the i -th thread being responsible for downloading the i -th coded chunk of each file. Each experiment lasted longer than 24 hours. We alternated between different settings to capture similar time of day characteristics across all settings.

The experiments are conducted within all 8 availability regions in Amazon S3. Except for the “US Standard” availability region, all other 7 regions demonstrate similar performance statistics that are consistent over different times and days. On the other hand, the performance of “US Standard” demonstrated significant variation even at different times in the same day, as illustrated in Fig.4(a) and Fig.4(b). We conjecture that the different and inconsistent behavior of “US Standard” might be due to the fact that it targets a slightly different usage pattern and it may employ a different implementation for

that reason⁶. We will exclude “US Standard” from subsequent discussions. For conciseness, we only show a limited subset of findings for availability region “North California” that are representative for regions other than “US Standard”:

(1) In both Unique Key and Shared Key, the task delay distribution observed by different threads are almost identical. The two approaches are indistinguishable even beyond 99.9th percentile. Fig.4(c) shows the complementary cumulative distribution function (CCDF) of task delays observed by individual threads for 1MB chunks and $n = 4$. Both approaches demonstrate large delay spread in all regions.

(2) Task delays for different threads in Unique Key show close to zero correlation, while they demonstrate slightly higher correlation in Shared Key, as it is expected. With all different settings, the cross correlation coefficient between different threads stays below 0.05 in Unique Key and ranges from 0.11 to 0.17 in Shared Key. Both approaches achieve significant service delay improvements. Fig.5 plots the CCDF of service delays for downloading 3MB files with 1MB chunks ($k = 3$) with $n = 3 \sim 6$, assuming all n tasks in a batch start at the same time. In this setting, both approaches reduce the 99th percentile delays by roughly 50%, 65% and 80% by downloading 1, 2 and 3 extra coded chunks. Although Shared Key demonstrates up to 3 times higher cross correlation coefficient, there is no meaningful statistical distinction in

⁶See http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

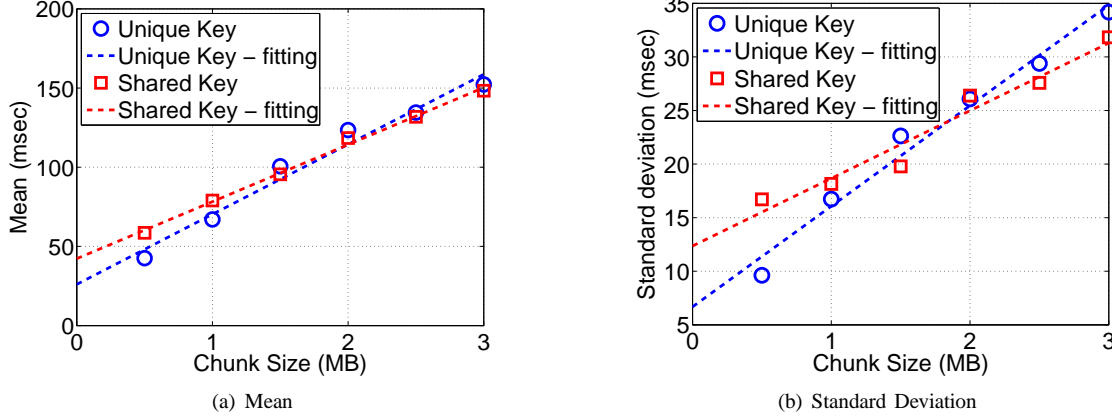


Fig. 6. Delay statistics vs. chunk size

service delay between the two approaches until beyond the 99th percentile. All availability regions experience different degrees of degradation at high percentiles with Shared Key due to the higher correlation. Significant degradation emerges from around 99.9th percentile and beyond in all regions except for “Sao Paulo”, in which degradation appears around 99th percentile.

(3) Task delays are always lower bounded by some constant $\Delta \geq 0$ that grows roughly linearly as chunk size increases. This constant part of delay cannot be reduced by using more threads: see the flat segment at the beginning of the CCDF curves in Fig.4 and Fig.5. Since this constant portion of task delays is unavoidable, it leads to the negative effect of using larger n since there is a minimum cost of system resource of $n\Delta$ (time \times thread) that grows linearly in n . This cost leads to a reduced capacity region for using more redundant tasks, as illustrated in the example of Fig.1. We observe that the two approaches deliver almost identical total delays (queueing + service) for all arrival rates, in spite of the degraded service delay with Shared Key at very high percentile. So we only plot the results with Shared Key in Fig.1.

(4) Both the mean and standard deviation of task delays grow roughly linearly as chunk size increases. Fig.6 plots the measured mean and standard deviation of task delays in both approaches at different chunk sizes. Also plotted in the figures are least squares fitted lines for the measurement results. As the figures show, performance of Unique Key and Shared Key are comparable also in terms of how delay statistics scale as functions of the chunk size. Notice that the extrapolations at chunk size = 0 are all greater than zero. We believe this observation reflects the costs of non-I/O-related operations in the storage cloud that do not scale proportionally to object size: for example, the cost to locate the requested object. We also believe such costs contribute partially to the minimum task delay constant Δ .

SUMMARY: Our measurement study shows that dynamic chunking while preserving weak correlation across different chunks is realizable through both Unique Key and Shared Key. We believe Shared Key is a reasonable choice for implementing dynamic chunking given that it is able to deliver

delay performance comparable to Unique Key at a much lower cost of storage capacity. We turn our attention on how to pick the best choices of chunking and FEC rate in the remaining parts of the paper.

C. Model of Task Delays

For the analysis present in the next section, we model the task delays as independently distributed random variables whose mean and standard deviation grow linearly as chunk size B increases. More specifically, we assume the task delay D_t for chunk size B following distribution in the form of

$$D_t(B) \sim \Delta(B) + \exp(\mu(B)), \quad (1)$$

where $\Delta(B) = \bar{\Delta} + \tilde{\Delta}B$ captures the lower bound of task delays as in observation (3), and $\exp(\mu(B))$ represents a exponential random variable that models the tail of the CCDF. The mean and standard deviation of the exponential tail both equal to $\frac{1}{\mu(B)} = \bar{\Psi} + \tilde{\Psi}B$. With this model, constants $\bar{\Delta}$ and $\bar{\Psi}$ together capture the non-zero extrapolations of the mean and standard deviation of task delays at chunk size 0, and similarly, constants $\tilde{\Delta}$ and $\tilde{\Psi}$ together capture the rate at which the mean and standard deviation grow as chunk size increases, as in observation (4).

IV. DESIGN OF TOFEC

For the analysis in this section, we group requests into classes according to the tuple (type, size). Here type can be read or write, and can potentially be other type of operations supported by the cloud storage. Each type of operation has its own set of delay parameters $\{\bar{\Delta}, \tilde{\Delta}, \bar{\Psi}, \tilde{\Psi}\}$. Subscripts will be used to indicate variables associated with each class. We use n_i , k_i and r_i to denote the code length, dimension and redundancy ratio for the code used to serve class i requests. Also let p_i denote the fraction of total arrivals contributed by class i . We use vectors \hat{n} , \hat{k} , \hat{r} and \hat{p} to denote the collection of corresponding variables for all classes.

The system throughput is defined as the average number of successfully served requests per unit time. The **static code capacity** $C_{sta}(\hat{p}, \hat{k}, \hat{r})$ is defined as the maximum deliverable throughput, assuming \hat{p} , \hat{k} , and \hat{r} are fixed. The **full capacity**

$C(\hat{p})$ is then defined as the maximum static code capacity considering all possible choices of (\hat{k}, \hat{r}) with \hat{p} fixed. For a given request arrivals rate λ , the system throughput equals to the smaller of λ and the (static or full) capacity.

A. Problem Formulation and Main Result for Static Strategy

Given total arrival rate λ and composition of requests \hat{p} , we want to find the best choice of FEC code for each class such that the average expected total delay is minimized. Relaxing the requirement for n_i and k_i being integers, this is formulated as the following minimization problem⁷:

$$\begin{aligned} \min_{\hat{k}, \hat{r}} \quad & D_q + \sum_i p_i D_{s,i} \quad (*) \\ \text{s.t.} \quad & k_i \geq 0, \quad r_i \geq 1 \quad \forall i, \\ & \lambda < C_{sta}(\hat{p}, \hat{k}, \hat{r}). \end{aligned}$$

In the above formulation, we use \hat{k} and \hat{r} as the optimizing variables, instead of a more intuitive choice of \hat{n} and \hat{k} . This choice helps simplify the analysis because \hat{k} and \hat{r} can be treated as independent variables while \hat{n} being subject to the constraint $\hat{n} \geq \hat{k}$. In subsequent sections, we first introduce approximations for the expected queueing and service delays assuming that the FEC code used to serve requests of each class is predetermined and fixed (Section IV-B). Then we show that optimal solutions to the above non-convex optimization problem exhibit the following property (Section IV-C):

The optimal values of n_i , k_i and r_i can all be expressed as functions solely determined by Q – the expected length of the request queue:

$$n_i = N_i(Q), \quad k_i = K_i(Q) \quad \text{and} \quad r_i = R_i(Q).$$

N_i , K_i and R_i are all strictly decreasing functions of Q .

This finding is then used as the guideline in the design of our backlog-driven adaptive strategy TOFEC (Section IV-D).

B. Approximated Analysis of Static Strategies

Denote J_i as the file size of class i . Consider a request of class i served with an (n_i, k_i) MDS code, i.e., $B_i = J_i/k_i$. First suppose all n_i tasks start at the same time, i.e., $T_1 = T_{n_i}$. In this case, given our model for task delays, it is trivial to show that the expected service delay equals to

$$\begin{aligned} D_{s,i} &= \Delta_i(J_i/k_i) + \frac{1}{\mu_i(J_i/k_i)} \sum_{j=n_i-k_i+1}^{n_i} \frac{1}{j} \\ &\approx \Delta_i(J_i/k_i) + \frac{1}{\mu_i(J_i/k_i)} \ln \left(\frac{n_i}{n_i - k_i} \right) \\ &= \bar{\Delta}_i + \frac{\tilde{\Delta}_i J_i}{k_i} + \left(\bar{\Psi}_i + \frac{\tilde{\Psi}_i J_i}{k_i} \right) \ln \left(\frac{r_i}{r_i - 1} \right). \quad (2) \end{aligned}$$

For the analysis, we approximate the summation $\sum_{j=n_i-k_i+1}^{n_i} 1/j$ with its integral upper bound

⁷Notice that all classes share the same queueing delay. Also, we require $k_i \geq 0$ instead of $k_i \geq 1$ for a technicality to simplify the proof of the uniqueness of the optimal solution. We require $r_i \geq 1$ since $n_i \geq k_i$. $\lambda < C_{sta}(\hat{p}, \hat{k}, \hat{r})$ is imposed for queue stability.

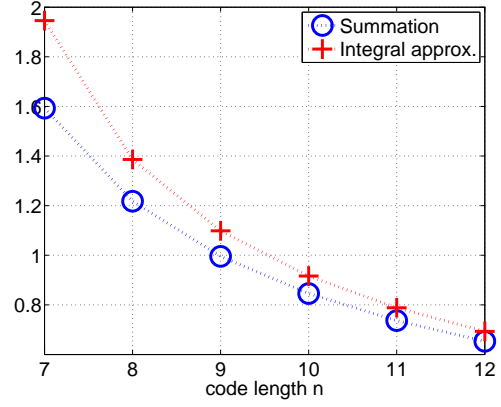


Fig. 7. Comparison of the summation term in Eq.2 and integral approximation for $k = 6$.

$\int_{n_i-k_i}^{n_i} \frac{1}{x} dx = \ln \left(\frac{n_i}{n_i - k_i} \right)$. The gap of approximation is always upper bounded by the Euler-Mascheroni constant ≈ 0.577 for any $n_i - k_i \geq 1$ and quickly diminishes to 0 when n_i gets large, as illustrated in Fig.7. Although the gap goes to ∞ as $n_i - k_i \rightarrow 0$, it does not really matter for the purpose of this paper since any optimal solution with n_i closer to k_i only means we should set $n_i = k_i$.

Also define the system usage (or simply “cost”) of a request as the sum of the amount of time each of its tasks being served by a thread⁸. When all tasks start at the same time, its expected system usage is (see Section IV of [3] for detailed derivation)

$$\begin{aligned} U_i &= n_i \Delta_i(J_i/k_i) + \frac{k_i}{\mu_i(J_i/k_i)} \\ &= \bar{\Delta}_i k_i r_i + \tilde{\Delta}_i J_i r_i + \bar{\Psi}_i k_i + \tilde{\Psi}_i J_i. \quad (3) \end{aligned}$$

Given that class i contributes to p_i fraction of the total arrivals, the average cost per request is $\bar{U} = \sum_i p_i U_i$. With L simultaneously active threads, the departure rate of the system as well as the request queue is L/\bar{U} (request/unit time). In light of this observation, we approximate the request queue with an $M/M/1$ queue with service rate L/\bar{U} ⁹. In other words, the static code capacity for a given \hat{p} and fixed code choice (\hat{k}, \hat{r}) is approximated by

$$C_{sta}(\hat{p}, \hat{k}, \hat{r}) = \frac{L}{\bar{U}}. \quad (4)$$

Let

$$\begin{aligned} \bar{\lambda} &= \lambda \bar{U} \\ &= \lambda \sum_i p_i (\bar{\Delta}_i k_i r_i + \tilde{\Delta}_i J_i r_i + \bar{\Psi}_i k_i + \tilde{\Psi}_i J_i) \quad (5) \end{aligned}$$

represent the arrival rate of system usage imposed by the request arrivals. Then the last inequality constraint of the optimization problem (*) becomes

$$\bar{\lambda} < L. \quad (6)$$

⁸The time a task j being served is $D_{t,j}$ if it completes successfully; $(X_{(k)} - T_j)$ if it starts but is terminated preemptively; and 0 if it is canceled while waiting in the task queue.

⁹This $M/M/1$ approximation is a special case of the $M/G/1$ approximation used in [3]: $D_q = \frac{\beta \lambda \bar{U}^2}{2L(L - \lambda \bar{U})}$, with $\beta = 1$. Our findings in this paper readily generalizes to accommodate the $M/G/1$ approximation.

With the $M/M/1$ queue approximation, the queueing delay in the original system at total arrival rate λ is approximated by

$$D_q = \frac{1}{L/\bar{U} - \lambda} - \frac{1}{L/\bar{U}} = \frac{\lambda \bar{U}^2}{L(L - \lambda \bar{U})}. \quad (7)$$

Noticing that given \hat{p} , the (approximated) static coded capacity $C_{sta}(\hat{p}, \hat{k}, \hat{r}) = L/\bar{U}$ is maximized when $k_i = 1, r_i = 1, \forall i$, we approximate the full capacity $C(\hat{p}) = C_{sta}(\hat{p}, \mathbf{1}, \mathbf{1})$, where $\mathbf{1}$ denotes the all-one vector. We acknowledge that the above approximations are quite coarse, especially because tasks of the same batch do not start at the same time in general. However, remember that the main objective of this paper is to develop a practical solution that can achieve the optimal throughput-delay trade-off. According to the simulation results, these approximations are sufficiently good for this purpose.

C. Optimal Static Strategy

Even with the above approximations, the minimization problem (*) is not a convex optimization problem: the feasible region is not a convex set due to the $k_i r_i$ terms in $\bar{\lambda}$. In general, non-convex optimization problems are difficult to solve. Fortunately, we are able to prove the following theorem according to which this non-convex optimization problem can be solved numerically with great efficiency.

Theorem 1: The optimal solutions to (*) must satisfy the following equations, regardless of λ and \hat{p} .

$$k_i = \Omega_i(r_i) \triangleq \frac{\bar{\Delta}_i \Gamma_i - \tilde{\Psi}_i J_i + \sqrt{(\bar{\Delta}_i \Gamma_i - \tilde{\Psi}_i J_i)^2 + 4 \bar{\Psi}_i \tilde{\Delta}_i J_i \Gamma_i}}{2 \bar{\Psi}_i}, \quad \forall i \quad (8)$$

$$\frac{L(\bar{\Psi}_i k_i + \tilde{\Psi}_i J_i)}{k_i r_i (r_i - 1)(\bar{\Delta}_i k_i + \tilde{\Delta}_i J_i)} = \frac{L(\bar{\Psi}_j k_j + \tilde{\Psi}_j J_j)}{k_j r_j (r_j - 1)(\bar{\Delta}_j k_j + \tilde{\Delta}_j J_j)}, \quad \forall i, j \quad (9)$$

where $\Gamma_i = \frac{J_i r_i (r_i - 1)}{\bar{\Delta}_i r_i + \tilde{\Psi}_i} \left(\bar{\Delta}_i + \tilde{\Psi}_i \ln \frac{r_i}{r_i - 1} \right)$. Moreover, when λ and \hat{p} are given, the optimal solution is the unique solution to the above equations and the one below:

$$\left(\frac{L}{L - \bar{\lambda}} \right)^2 - 1 = \frac{L(\bar{\Psi}_i k_i + \tilde{\Psi}_i J_i)}{k_i r_i (r_i - 1)(\bar{\Delta}_i k_i + \tilde{\Delta}_i J_i)}, \quad \forall i. \quad (10)$$

Proof: See Appendix. ■

The importance of Theorem 1 is two-fold:

- 1) With m different classes of requests, the seemingly $2m$ -dimension optimization problem is in fact 1-dimensional: According to Eq.8, the optimal k_i is fully determined by the optimal r_i (vice versa). Moreover, according to Eq.9, the optimal r_i further fully determines the optimal choices of k_j and r_j for all other $j \neq i$. In other words, the knowledge of the optimal choice of

any r_i (or k_i) is sufficient to derive the complete optimal choice of (\hat{k}, \hat{r}) .

- 2) The optimal solution (n_i, k_i and r_i) is fully determined by $\bar{\lambda}$, hence it is *virtually* independent of the particular λ and \hat{p} : λ and \hat{p} appear in the above equations only in the form of $\bar{\lambda}$ in Eq.10. So for any two different pairs of (λ, \hat{p}) and (λ', \hat{p}') , as long as $\bar{\lambda} = \bar{\lambda}'$, they share **the same optimal choice of codes!** An implication of this is that the m -class optimization problem can be solved by solving a set of m independent single-class subproblems: the i -th subproblem solves for the optimal (k_i, r_i) with class- i -only arrivals at rate λ_i such that $\lambda_i U_i = \bar{\lambda}$, because it is equivalent to the m -class problem when $\lambda' = \bar{\lambda}/U_i$ and \hat{p}' such that $p'_i = 1$ and $p'_j = 0, \forall j \neq i$.

The second observation above is of particular interest to the purpose of this paper. It suggests that adaptation of different classes can be done separately, as if only arrivals are for the class under consideration. This significantly simplifies the design of our adaptive strategy TOFEC, resulting in great computational efficiency and flexibility.

D. Adaptive Strategy TOFEC

Despite being the mathematical foundation for the design of TOFEC, Theorem 1 at its current formulation is not very useful in practice. This is because the code adaptation is based on the knowledge of the total workload λ and the popularity distribution of different classes \hat{p} as per Theorem 1. In practice, both quantities usually demonstrate high degree of volatility, making accurate on-the-fly estimation quite difficult and/or unreliable. So in order to achieve effective code adaptation, a more robust system metric that is easy to measure with high accuracy is desirable.

Observe that the expected length of the request queue is

$$Q = \lambda D_q = \frac{(\lambda \bar{U})^2}{L(L - \lambda \bar{U})} = \frac{\bar{\lambda}^2}{L(L - \bar{\lambda})}, \quad (11)$$

which can be rewritten as

$$\bar{\lambda} = \frac{L \left(\sqrt{Q^2 + 4Q} - Q \right)}{2}. \quad (12)$$

It is trivial and intuitive that Q is a strictly increasing function of $\bar{\lambda}$ and vice versa. On the other hand, it is not hard to verify that optimal n_i, k_i and r_i are all strictly decreasing functions of $\bar{\lambda}$ according to Theorem 1. Replacing $\bar{\lambda}$ with Eq.12, we can conclude the following corollary:

Corollary 1: The optimal values of n_i, k_i and r_i can all be expressed as strictly decreasing functions of Q :

$$n_i = N_i(Q), \quad k_i = K_i(Q) \quad \text{and} \quad r_i = R_i(Q). \quad (13)$$

The findings of Corollary 1 conform to the following intuition:

- At light workload (small λ), there should be little backlog in the request queue (small Q) and the service delay dominates the total delay. In this case, the system is not operating in the capacity-limited regime. So it is beneficial to increase the level of chunking and redundancy (large k_i and r_i) to reduce delay.

Algorithm 1: TOFEC (Throughput Optimal FEC Cloud)

Initialization: $\bar{q} = 0$. When request arrives

- 1 $q \leftarrow$ queue length upon arrival of request;
 - 2 $i \leftarrow$ class that request belongs to;
 - 3 $\bar{q} \leftarrow \alpha \bar{q} + (1 - \alpha)q$;
 - 4 Find $k \leq k_i^{max}$ such that $\bar{q} \in [H_{i,k+1}^N, H_{i,k}^N)$;
 - 5 Find $n \leq n_i^{max}$ such that $\bar{q} \in [H_{i,n+1}^N, H_{i,n}^N)$;
 - 6 $n \leftarrow \min(r_i^{max} k, n)$;
 - 7 Serve request with an (n, k) code when it becomes HoL;
-

- At heavy workload (larger λ), there will be a large backlog in the request queue (large Q) and the queueing delay dominates the total delay. In this case, the system operates in the capacity-limited regime. So it is better to reduce the level of chunking and redundancy (small k_i and r_i) to support higher throughput.

More importantly, it suggests the sufficiency to choose the FEC code solely based on the length of the request queue – a very robust and easy to obtain system metric – instead of less reliable estimations of λ and \hat{p} . As will be discussed later, queue length has other advantages over arrival rate in a dynamic setting.

The basic idea of TOFEC is to choose $n_i = N_i(q)$ and $k_i = K_i(q)$ for a request of class i , where q is the queue length upon the arrival of the request. When this is done to all request arrivals to the system, it can be expected the average code lengths (dimensions) and expected queue length Q satisfy Eq.13, hence the optimal delay is achieved. In TOFEC, this is implemented with a threshold based algorithm, which can be performed very efficiently. For each class i , we first compute the expected queue length given $n_i \in \{1, \dots, n_i^{max}\}$ is the optimal code length by

$$Q_{i,n_i}^N = N_i^{-1}(n_i). \quad (14)$$

Here n_i^{max} is the maximum number of tasks allowed for a class i request. Since N_i is a strictly decreasing function, its inverse N_i^{-1} is a well-defined strictly decreasing function. As a result, we have $Q_{i,1}^N > Q_{i,2}^N > \dots > Q_{i,n_i^{max}}^N > 0$. Note that our goal is to use code length n if the queue length q is around $Q_{i,n}^N$, so we want a set of thresholds $\{H_{i,n}^N\}$ such that

$$H_{i,1}^N > Q_{i,1}^N > H_{i,2}^N > Q_{i,2}^N > \dots \\ \dots > H_{i,n_i^{max}}^N > Q_{i,n_i^{max}}^N > H_{i,n_i^{max}+1}^N = 0,$$

and will use n such that $q \in [H_{i,n+1}^N, H_{i,n}^N)$. In our current implementation of TOFEC, we use $H_{i,n}^N = (Q_{i,n}^N + Q_{i,n-1}^N) / 2$ for $n = 2, \dots, n_i^{max}$ and $H_{i,1}^N = \infty$. A set of thresholds $\{H_{i,k}^{K_i^{max}}\}$ for adaptation of k_i is found in a similar fashion.

The adaptation mechanism of TOFEC is summarized in pseudo-codes as Algorithm 1. Note that in Step 6 we reduce n to $r_i^{max} k$ if the redundancy ratio of the code chosen in the previous steps is higher than r_i^{max} – the maximum allowed redundancy ratio for class i . Also, instead of comparing q directly with the thresholds, we compare an exponential moving average $\bar{q} = \alpha \bar{q} + (1 - \alpha)q$, with a memory factor

$0 \leq \alpha \leq 1$, against the thresholds to determine n and k . The moving average is used to mitigate the transient variation in queue length so that n and k will not change too frequently. It is obvious that we only need to set $\alpha = 0$ in order to use instantaneous queue length q for the adaptation since in this case $\bar{q} = q$.

It is worth pointing out that TOFEC's threshold based adaptation is

- 1) Independent of \hat{p} : The thresholds for each class is computed a priori without any knowledge or assumption of \hat{p} . Once computed, the thresholds can be reused for all realizations of different \hat{p} , even if \hat{p} is time-varying;
- 2) Independent across classes: For a class i , computation of its thresholds require knowledge of neither the number nor the delay parameters of other classes. The adaptation of class i is also independent of those of the other classes.

These two properties of independence are direct result of the implication of Theorem 1 we discussed before. Thanks to these nice properties, it is very easy in TOFEC to add support for a new class in an incremental fashion: simply compute the thresholds for the new class, leaving the thresholds for the existing, say m , classes untouched. The old and new thresholds together will then produce the optimal choice of codes for the incremented set of $m + 1$ classes.

V. EVALUATION

We now demonstrate the benefits of TOFEC's adaptation mechanism. We evaluate TOFEC's adaptation strategy and show that it outperforms static strategies with both constant and changing workloads, as well as a simple greedy heuristic that will be introduced later.

A. Simulation Setup

We conducted trace-driven simulations for performance evaluation for both single-class and multi-class scenarios with both read and write requests of different file sizes. Due to lack of space, we only show results for the scenario with one class (read, 3MB). But we must emphasize that it is representative enough so that the findings to be discussed in this section are valid for other settings (different file sizes, write requests, and multiple classes). We assume that the system supports up to $L = 16$ simultaneously active threads. We set the maximum code dimension and redundancy ratio to be $k^{max} = 6$ and $r^{max} = 2$, because we observe negligible gain in service delay beyond this chunking and redundancy level from our measurements. We use traces collected in May and June 2013 in availability region "North California". In order to compute the thresholds for TOFEC, we need estimations of the delay parameters $\{\bar{\Delta}, \hat{\Delta}, \bar{\Psi}, \hat{\Psi}\}$. For this, we first filter out the worst 10% task delays in the traces, then we compute the delay parameters from the least squares linear approximation for the mean and standard deviation of the remaining task delays. We use memory factor $\alpha = 0.99$ in TOFEC.

In addition to the static strategies, we develop a simple *Greedy* heuristic strategy for the purpose of comparison. Unlike the adaptive strategy in TOFEC, Greedy does not

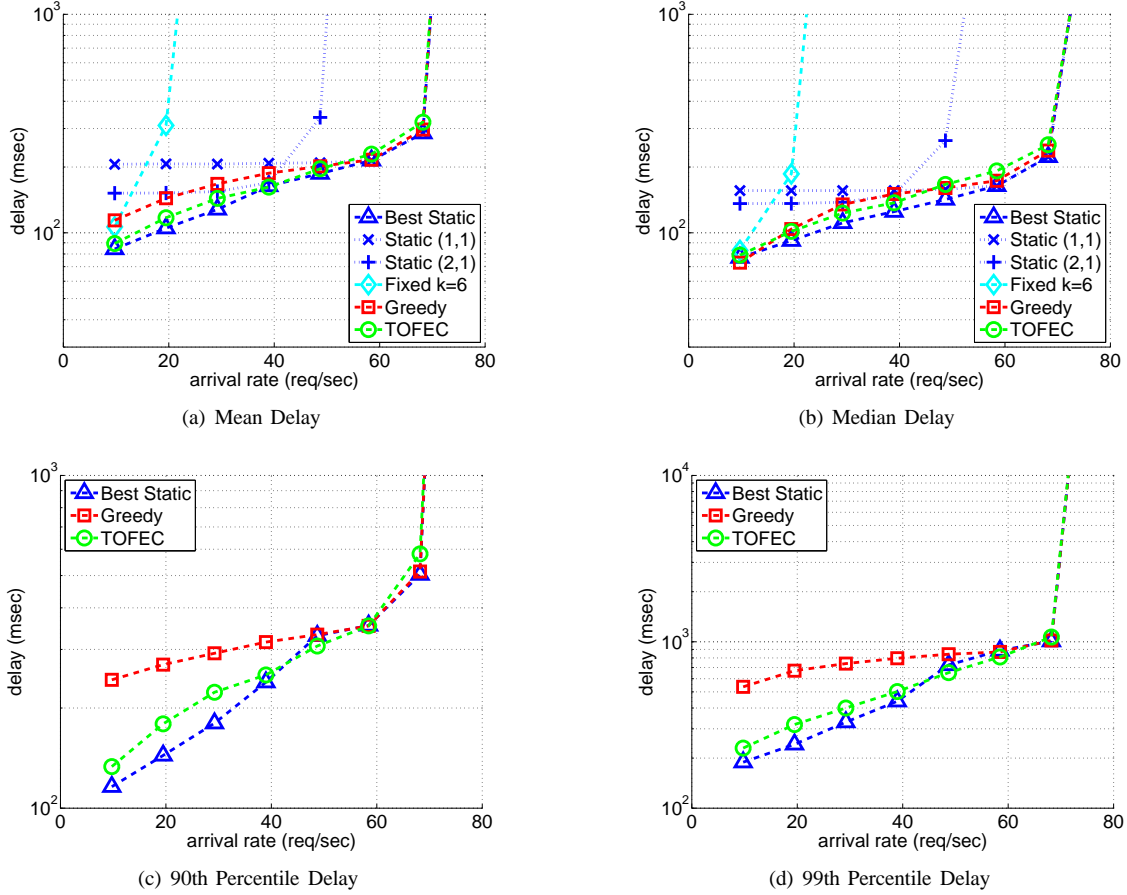


Fig. 8. Delay performance in read only scenario

require prior-knowledge of the distribution of task delays, yet as the results will reveal, it achieves a competitive mean delay performance. In Greedy, the code to be used to serve a request in class i is determined by the number of idle threads upon its arrival: suppose there are l idle threads, then

$$k_i = \begin{cases} 1, & \text{if } l = 0, \\ \min(k_i^{max}, l), & \text{otherwise;} \end{cases}$$

and similarly

$$n_i = \begin{cases} 1, & \text{if } l = 0, \\ \min(r_i^{max} k_i, l), & \text{otherwise.} \end{cases}$$

The idea of Greedy is to first maximize the level of chunking with the idle threads available, then increase the redundancy ratio as long as there are idle threads remaining.

B. Throughput-Delay Trade-Off

Fig.8 shows the mean, median, 90th percentile and 99th percentile delays of TOFEC and Greedy with Poisson arrivals at different arrival rates of λ . We also run simulations with static strategies for all possible combinations of (n, k) at every arrival rate. In a brute-force fashion, we find the best mean, median, 90th and 99th percentile delays achieved with static strategies and use them as the baseline. Fig.8(a) and Fig.8(b) also plot the mean and median delay performance of the basic

static strategy with no chunking and no replication, i.e., $(1, 1)$ code; the simple replication static strategy with a $(2, 1)$ code; and the backlog-based adaptive strategy from [3] with fixed code dimension $k = 6$ and $n \leq 12$.

As we can see, both TOFEC and Greedy successfully support the full capacity region – the one supported by basic static – while achieving almost optimal mean and median delays throughout the full capacity region. At light workload, TOFEC delivers about $2.5\times$ improvement in mean delay when compared with the basic static strategy, and about $2\times$ when compared with simple replication (from 205ms and 151ms to 84ms). It also reduces the median delay by about $2\times$ from that of basic and simple replication (from 156ms and 138ms to 74ms). Meanwhile Greedy achieves about $2\times$ improvement in both mean (89ms) and median delays (79ms) over basic.

With heavier workload, both TOFEC and Greedy successfully adapt their codes to keep track with the best static strategies, in terms of mean and median delays. It is clear from the figures that both TOFEC and Greedy achieve our primary goal of retaining full system capacity, as supported by basic static strategy. On the contrary, although simple replication has slightly better mean and median delays than basic under light workload, it fails to support arrival rates beyond 70% of the capacity of basic. Meanwhile, the adaptive strategy from [3] with fixed code dimension $k = 6$ can only support less than 30% of the original capacity region, although it achieves the

best delay at very light workload.

While the two adaptive strategies have similar performance in mean and median, TOFEC outperforms Greedy significantly at high percentiles. As Fig.8(c) and Fig.8(d) demonstrate, TOFEC is on a par with the best static strategies at the 90th and 99th percentile delays throughout the whole capacity region. On the other hand, Greedy fails to keep track of the best static performance at lower arrival rates. At light workload, TOFEC's is over $2\times$ and $2.5\times$ better than Greedy at the 90th and 99th percentiles. Less interesting is the case with heavy workload when the system is capacity-limited. Hence both strategies converge to the basic static strategy using mostly (1, 1) code, which is optimal at this regime.

C. Delay Variation and Choice of Codes

We further compare the standard deviation (STD) of TOFEC, Greedy and the best static strategy. STD is a very important performance metric because it directly relates to whether customers can receive consistent QoS. In certain applications, such as video streaming, maintaining low STD in delay can be even more critical than achieving low mean delay. As we can see in Fig.9, for the region of interest with light to medium workload, TOFEC delivers $2\times$ to $3\times$ lower STD than Greedy does. Moreover, in spite of its dynamic adaptive nature, TOFEC in fact matches with the best static strategy very well throughout the full capacity region. This suggests the code choice in TOFEC indeed converges to the optimal.

The convergence to optimal becomes more obvious when we look into the fraction of requests served by each choice of code. In Fig.10 we plot the compositions of requests served by different code dimension k 's. At each arrival rate, the two bars represent TOFEC and Greedy. For each bar, blocks in different colors represent the fraction of requests served with code dimension 1 through 6, from bottom to top. TOFEC's choice of k demonstrates a high concentration around the optimal value: at all arrival rate, over 80% requests are served by 2 neighboring values of k around the optimal, and this fraction quickly diminishes to 0 for codes further from the optimal. Moreover, as arrival rate varies from low to high, TOFEC's choice of k transitions quite smoothly as $(5, 6) \rightarrow (3, 4) \rightarrow (2, 3) \rightarrow (1, 2)$ and eventually converges to a single value 1 as workload approaches system capacity.

On the contrary, Greedy tends to round-robin across all possible choices of k and majority of requests are served by either $k = 1$ or 6. So Greedy is effectively alternating between the two extremes of no chunking and very high chunking, instead of staying around the optimal. Such "all or nothing" behavior results in the $2\times$ to $3\times$ worse STD shown in Fig.9. So TOFEC provides much better QoS guarantee.

D. Adapting to Changing Workload

We further examine how well the two adaptive strategies adjust to changes in workload. In Fig.11 we plot the total delay experienced by requests arriving at different times within a 600-second period, as well as the choice of code in the same period. The 600 seconds is divided into 3 phases, each lasts 200 seconds. The arrival rate is 10 request/second in phases

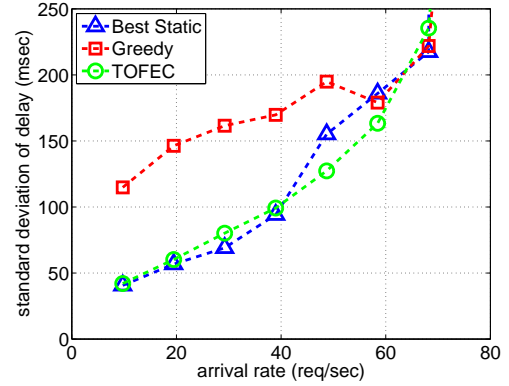


Fig. 9. Comparison of standard deviation

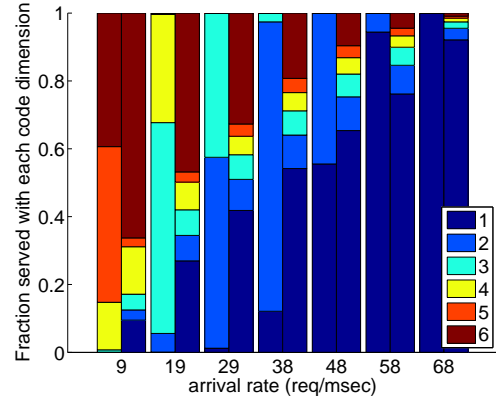


Fig. 10. Composition of k . Left: TOFEC, Right: Greedy

1 and 3, and 80 request/second (slightly $> C$) in phase 2. The corresponding optimal choices of codes (n, k) are (10, 5) for phases 1 and 3, and (1, 1) for phase 2. For the purpose of comparison, we also implement an "Ideal" rate-driven strategy that has perfect knowledge of the arrival rate of each phase and picks the optimal code accordingly as the baseline. We can see that both TOFEC and Greedy are quite agile to changes in arrival rate and quickly converge to a good composition of codes that delivers optimal mean delays within each phase, comparable to that of Ideal.

From Fig.11(b) we can further observe that TOFEC is especially responsive in face of workload surge (from phase 1 to 2). This is because the suddenly increased arrival rate immediately builds up a large backlog, which in turn forces TOFEC to pick a code with the smallest $k = 1$. When the arrival drops (from phase 2 to 3), instead of immediately switching back to codes with $k = 5$, TOFEC gradually transitions to the optimal value of $k = 5$. Such "smoothing" behavior when workload reduces is actually beneficial. This is because the request queue has been built up during the preceding period of heavy workload. So, if k is set to 5 right after arrival rate drops, it will produce a throughput so low that it takes a much longer time to reduce the queue length to the desired level, and requests arrive during this period will suffer from long queueing delay even though they are being served with the optimal code. On the other hand, TOFEC's queue length driven adaptation sticks with smaller k , which

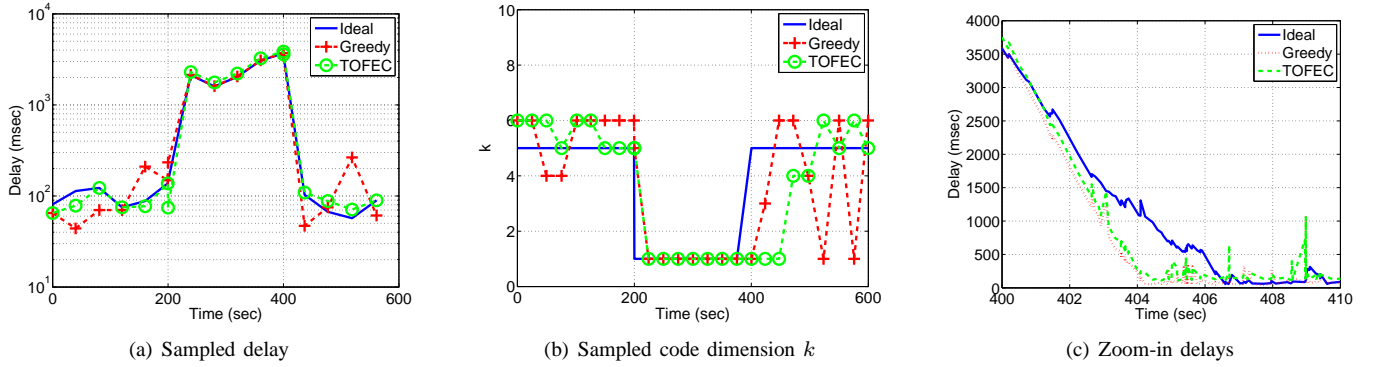


Fig. 11. Adaptation to changing workload

delivers higher throughput, to drain the queue much faster to the desired level. As we can see in Fig.11(c), which plots the delay traces for requests arrive in the first 10 seconds of phase 3, TOFEC and Greedy both reduce their delay to optimal almost $1.8\times$ faster than Ideal does after workload decreases. This is another advantage of using queue length instead of arrival rate to drive code adaptation.

We can also see that TOFEC's choice of code is much more stable than that of Greedy. While TOFEC shows little variation around the optimal in each phase, Greedy keeps oscillating between $k = 1$ and $k = 6$ when the optimal is 1! This is consistent with the “all or nothing” behavior of Greedy observed in Fig.10.

VI. RELATED WORK

FEC in connection with multiple paths and/or multiple servers is a well investigated topic in the literature [8], [9], [10], [11]. However, there is very little attention devoted to the queueing delays. FEC in the context of network coding or coded scheduling has also been a popular topic from the perspectives of throughput (or network utility) maximization and throughput vs. service delay trade-offs [12], [13], [14], [15]. Although some incorporate queueing delay analysis, the treatment is largely for broadcast wireless channels with quite different system characteristics and constraints. FEC has also been extensively studied in the context of distributed storage from the points of high durability and availability while attaining high storage efficiency [16], [17], [18].

Authors of [4] conducted theoretical study of cloud storage systems using FEC in a similar fashion as we did in our work [3]. Given that exact mathematical analysis of the general case is very difficult, authors of [4] considered a very simple case with a fixed code of $k = 2$ tasks. Shah et al. [5] generalize the results from [4] to $k > 2$. Both works rely on the assumption of exponential task delays, which hardly captures the reality. Therefore, some of their theoretical results cannot be applied in practice. For example, under the assumption of exponential task delays, Shah et al. have proved that it is optimal to always use the largest n possible throughout the full capacity region C , contradicting with simulation results using real-world measurements in [3] and this paper.

VII. CONCLUSION

This paper presents the first set of solutions for achieving the optimal throughput-delay trade-off for scalable key-value storage access using erasure codes with variable chunk sizing and rate adaptation. We establish the viability of this approach through extensive measurement study over the popular public cloud storage service Amazon S3. We develop two adaptation strategies: TOFEC and Greedy. TOFEC monitors the local backlog and compares it against a set of thresholds to dynamically determine the optimal code length and dimension. Our trace-driven simulation shows that TOFEC is on a par with the best static strategy in terms of mean, median, 90th, and 99th percentile delays, as well as delay variation. To compute the thresholds, TOFEC requires knowledge of the mean and variance of cloud storage access delays, which is usually obtained by maintaining a log of delay traces. On the other hand, Greedy does not require any knowledge of the delay profile or logging but is able to achieve mean and median delays comparable to those of TOFEC. However, it falls short in important QoS metrics such as higher percentile delays and variation. It is part of our ongoing work to develop a strategy that matches TOFEC's high percentile delay performance without prior knowledge and logging.

REFERENCES

- [1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure Coding in Windows Azure Storage,” in *USENIX ATC*, 2012.
- [2] S. L. Garfinkel, “An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS,” Harvard University, Tech. Rep., 2007.
- [3] G. Liang and U. C. Kozat, “FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding,” *IEEE/ACM Trans. Networking*, preprint, 13 Nov. 2013, doi: 10.1109/TNET.2013.2289382.
- [4] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, “Codes Can Reduce Queueing Delay in Data Centers,” in *IEEE ISIT*, 2012.
- [5] N. B. Shah, K. Lee, and K. Ramchandran, “The MDS Queue: Analysing Latency Performance of Codes and Redundant Requests,” *arXiv:1211.5405*, Apr. 2013.
- [6] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, “Stout: An Adaptive Interface to Scalable Cloud Storage,” in *USENIX ATC*, 2010.
- [7] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzyva: Speculative Byzantine fault tolerance,” *ACM Transactions on Computer Systems*, vol. 27, pp. 7:1–7:39, 2010.
- [8] V. Sharma, S. Kalyanaraman, K. Kar, K. K. Ramakrishnan, and V. Subramanian, “MPLOT: A Transport Protocol Exploiting Multipath Diversity Using Erasure Codes,” in *IEEE INFOCOM*, 2008.

- [9] E. Gabrielyan, "Fault-Tolerant Real-Time Streaming with FEC thanks to Capillary MultiPath Routing," *Computing Research Repository*, 2006.
- [10] J. W. Byers, M. Luby, and M. Mitzenmacher, "Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads," in *IEEE INFOCOM*, 1999.
- [11] R. Saad, A. Serhrouchni, Y. Begliche, and K. Chen, "Evaluating Forward Error Correction performance in BitTorrent protocol," in *IEEE LCN*, 2010.
- [12] A. Eryilmaz, A. Ozdaglar, M. Medard, and E. Ahmed, "On the Delay and Throughput Gains of Coding in Unreliable Networks," *IEEE Trans. Inf. Theor.*, 2008.
- [13] W.-L. Yeow, A. T. Hoang, and C.-K. Tham, "Minimizing Delay for Multicast-Streaming in Wireless Networks with Network Coding," in *IEEE INFOCOM*, 2009.
- [14] T. K. Dikaliotis, A. G. Dimakis, T. Ho, and M. Effros, "On the Delay of Network Coding over Line Networks," *Computing Research Repository*, 2009.
- [15] U. C. Kozat, "On the Throughput Capacity of Opportunistic Multicasting with Erasure Codes," in *IEEE INFOCOM*, 2008.
- [16] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. Inf. Theor.*, 2010.
- [17] R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication," in *4th International Workshop, IPTPS*, 2005.
- [18] J. Li, S. Yang, X. Wang, and B. Li, "Tree-Structured Data Regeneration in Distributed Storage Systems with Regenerating Codes," in *IEEE INFOCOM*, 2010.

APPENDIX PROOF OF THEOREM 1

Proof: It is easy to verify that the objective function (*) is continuous and differentiable everywhere within the feasible region and the partial derivatives are

$$\frac{\partial(*)}{\partial k_i} = \left(\frac{L}{(L - \lambda \bar{U})^2} - \frac{1}{L} \right) p_i(\bar{\Delta}_i r_i + \bar{\Psi}_i) - p_i \frac{J_i}{k_i^2} \left(\bar{\Delta}_i + \bar{\Psi}_i \ln \frac{r_i}{r_i - 1} \right) \quad (15)$$

and

$$\frac{\partial(*)}{\partial r_i} = \left(\frac{L}{(L - \lambda \bar{U})^2} - \frac{1}{L} \right) p_i(\bar{\Delta}_i k_i + \bar{\Delta}_i J_i) - p_i \left(\bar{\Psi}_i + \frac{\bar{\Psi}_i J_i}{k_i} \right) \frac{1}{r_i(r_i - 1)} \quad (16)$$

Notice that for the whole the feasible region including the boundary, (*) is always lower bounded by 0. So there must exist at least one global optimal solution (\hat{k}^*, \hat{r}^*) that minimizes (*). Moreover, (*) goes to ∞ if and only if the operating point (\hat{k}, \hat{r}) approaches the boundary, i.e., $k_i \rightarrow 0$, or $r_i \rightarrow 1$, or $\lambda \rightarrow C_{sta}(\hat{p}, \hat{k}, \hat{r})$. Since (*) is ∞ for the whole boundary, the global optimal (\hat{k}^*, \hat{r}^*) must reside strictly within the feasible region. As a result, both Eq.15 and Eq.16 must evaluate to 0 at (\hat{k}^*, \hat{r}^*) . In the subsequent discussion, we prove that Eq.15 = 0 and Eq.16 = 0 has a unique solution within the feasible region. As a result, (\hat{k}^*, \hat{r}^*) equals to this solution and is also unique.

From Eq.16 = 0 we have:

$$\left(\frac{L}{(L - \lambda \bar{U})^2} - \frac{1}{L} \right) = \frac{\bar{\Psi}_i k_i + \bar{\Psi}_i J_i}{k_i r_i (r_i - 1) (\bar{\Delta}_i k_i + \bar{\Delta}_i J_i)}. \quad (17)$$

Plugging the above into Eq.15, we have:

$$\frac{k_i(\bar{\Psi}_i k_i + \bar{\Psi}_i J_i)}{\bar{\Delta}_i k_i + \bar{\Delta}_i J_i} = \frac{J_i r_i (r_i - 1)}{\bar{\Delta}_i r_i + \bar{\Psi}_i} \left(\bar{\Delta}_i + \bar{\Psi}_i \ln \frac{r_i}{r_i - 1} \right). \quad (18)$$

Notice that if r_i is fixed, Eq.18 is in fact a quadratic equation of k_i . Let $\Gamma_i(r_i)$ (or Γ_i for short) be the right hand side of Eq.18. Then we have

$$k_i = \frac{\bar{\Delta}_i \Gamma_i - \bar{\Psi}_i J_i + \sqrt{(\bar{\Delta}_i \Gamma_i - \bar{\Psi}_i J_i)^2 + 4 \bar{\Psi}_i \bar{\Delta}_i J_i \Gamma_i}}{2 \bar{\Psi}_i} \quad (19)$$

$$\triangleq \Omega_i(r_i). \quad (20)$$

We do not consider the other solution to Eq.18 because it is always ≤ 0 . It is easy to verify that Ω_i is a strictly increasing function of r_i within the feasible region.

Substituting k_i with $\Omega_i(r_i)$, the right hand side of Eq.17 can be written as a function $\pi_i(r_i)$, which can be shown to be strictly decreasing. Notice that Eq.17 must be satisfied for all i and the left hand side remains unchanged. Then

$$\pi_i(r_i) = \pi_j(r_j), \quad \forall i, j. \quad (21)$$

Note that π_i and π_j are strictly decreasing functions of r_i and r_j , respectively. This means that there is a one-to-one mapping between any r_i and r_j at the optimal solutions, and r_j is a strictly increasing function of r_i , namely $r_j = \Upsilon_{j,i}(r_i)$.

Now with $r_j = \Upsilon_{j,i}(r_i)$ and $k_j = \Omega_j(r_j) = \Omega_j(\Upsilon_{j,i}(r_i))$, Eq.17 becomes a equation that contains only one variable r_i . It is then not hard to show that for any given λ and \hat{p} , the left hand side of Eq.17 is a strictly increasing function of r_i , while the right hand side is $\pi_i(r_i)$, which is strictly decreasing. As a result, these two functions can equal for at most one value of r_i . In other words, equations Eq.18 and Eq.17 have at most one solution. The existence of a solution to these equations is guaranteed by the existence of (\hat{k}^*, \hat{r}^*) , so it must be unique. This completes the proof. ■



Guanfeng Liang (S'06-M'12) received his B.E. degree from University of Science and Technology of China, Hefei, Anhui, China, in 2004, M.A.Sc. degree in Electrical and Computer Engineering from University of Toronto, Canada, in 2007, and Ph.D. degree in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign, in 2012. He currently works with DOCOMO Innovations (formerly DOCOMO USA Labs), Palo Alto, CA, as a Research Engineer.



Ulas C. Kozat (S97-M04-SM10) received his B.Sc. degree in Electrical and Electronics Engineering from Bilkent University, Ankara, Turkey, in 1997, M.Sc. degree in Electrical Engineering from the George Washington University, Washington, DC, in 1999, and Ph.D. degree in Electrical and Computer Engineering from the University of Maryland, College Park, in 2004. He currently works at DOCOMO Innovations (formerly DOCOMO USA Labs), Palo Alto, CA, as a Principal Researcher.